School of Engineering

Australian National University

ENGN3213 / ENGN6213
Digital Systems & Microprocessors

Reverse Polish Calculator Project

V1.1

Copyright 2009-2012 ANU College of Engineering and Computer Science

Contents

1 Introduction **4**

2 Reverse Polish Notation **5**

2.1 History of Reverse Polish 5

2.2 Reverse Polish Notation 6

2.3 The Memory Device in Reverse Polish 7

2.4 The HP-35 Reversal Polish Algorithm 10

3 RPC Hardware Description **12**

3.1 General 12

3.1.1 Overview 12

3.2 Spartan 3E Board Peripherals 13

3.3 Switch Debouncing 13

3.3.1 Basic Character encodings and Communications 15

3.3.2 Serial Communications 17

3.3.3 Character Display in the Terminal Emulator 17

3.3.4 Possible Architectures for the RPN Engine 18

3.3.5 Important Advice 20

4 Project Requirements **22**

4.1 Calculator Reset 22

4.2 Implementation Levels of the RP Engine 22

4.2.1 RP Engine Level 1 23

4.2.2 RP Engine Level 2 24

4.2.3 RP Engine Level 3 26

4.3 Assessment 26

4.3.1 The Mid-Semester Design Report 27

4.3.2 The Mid-Semester Implementation of the Serial Echo Terminal . . . 28

4.3.3 UCF File 30

4.3.4 The End-of-Semester Hardware Tests 31

4.3.5 The RPC Manual 33

4.4 Project Rules 33

A Description of the MU0 Microprocessor **35**

A.1 Introduction 35

A.2 The Control Path Finite State Machine 37

A.3 MU0 in action 38

A.4 Running a Program on MU0 40

A.5 MU0 Assembly Language? 41

B Frequently Asked Questions **43**

1 Introduction

This project will give you the opportunity to design a system of modest complexity: a **reverse polish calculator** with 4 significant decimal digits. The project has various milestones among the specifications to allow you to do a top-down design and to tackle the project at various levels of complexity with plenty of scope for individual creativity. A major aspect of the project will be to explore different approaches to hardware block design in order to trade size and speed.

In this course I emphasise the **Register Transfer Level** (RTL) description of complex digital systems. One example of this approach is the design and implementation of the MU0 microprocessor. You should familiarise yourself with the details of the operation of MU0 and apply a similar approach to the present problem. This project also involves an interfacing component. We will look at communications between the FPGA and a PC using RS232 serial communications.

Additional information can be found on WATTLE

2 Reverse Polish Notation

2.1 History of Reverse Polish

Reverse polish notation or RPN is an arithmetic notation introduced by the Polish mathemetician Jan Lukasiewicz in 1920. During the 1960s and 1970s, RPN was widely used in scientific calculators. The Hewlett-Packard HP-35, shown in Figure 1 was the world's first handheld scientific calculator (1972) and was based on RPN ([**?**]).

Figure 1: The HP 35 calculator.

The arrival of the HP-35 was a significant event given the market dominance of slide rules and mechanical calculators for engineering computations. The HP-35 used a traditional floating decimal display that automatically switched to scientific notation. The fifteen digit LED display was capable of displaying a 10 digit mantissa plus its sign and a decimal point and a two digit exponent plus its sign. The display was unique in that the multiplexing was designed to illuminate a single LED segment at a time, rather than a single LED digit, because HP research had shown that this method was perceived by the human eye as brighter for equivalent power.

Architecturally, the calculator was a bit-serial machine that processed 56-bit floating-point numbers, representing 14-digit BCD (Binary Coded Decimal) numbers. Figure 2 shows the main board of the HP-35. As you can see, integrated dual in-line was the technology of the day.

Figure 2: The HP 35 main board.

2.2 Reverse Polish Notation

RPN is a simpler and more practical alternative to the conventional procedure for performing arithmetic calculations that we learned in school. The latter method is reliant on the use of parentheses and equals signs and is sometimes referred to as **infix** notation. RPN is also referred to as **postfix** notation.

RPN is easiest to explain by example. Consider the following simple operation,

4 + 5 =

In RPN this expression is written,

4 ENTER 5 +

There is just one operation key referred to as "ENTER". Computations are performed incrementally and results are stored in memory as we proceed. Here is a more complex example,

$(4 + 5 \times 2) / 7 =$

In RP we would do,

4 ENTER 5 ENTER 2 $\times$ + ENTER 7 /

Note the logical manner in which the calculation proceeds and how parentheses and equals signs are eliminated. **To do the project, you will need to familiarise yourself with Reverse Polish notation.**

2.3 The Memory Device in Reverse Polish

RP calculations require some form of limited memory to store variables and results. The RP algorithm is suited to a special memory device referred to as a **stack**. A **stack** is a computer term for a memory in which data is stored on a pile of registers. A stack is analogous to a filing system in which the latest document to be filed is placed on top of the document pile.

A stack is a **Last In First Out** (LIFO) memory. When a variable is stored, it is **pushed** onto the stack. When a variable is to be retrieved, variables higher on the stack have to be **popped** until we reach the desired variable. The stack does not need to be very deep i.e. have many memory levels. The HP 35 stack has only four levels.

To see how the stack would be used in RP, consider the following examples,

Example 1

$(4 + 2 \times 5) / (1 + 2 \times 3)$

In RPN this is described by,

4 ENTER 2 ENTER 5 × + ENTER 1 ENTER 2 ENTER 3 × + /

In the following table the S1-S4 refer to the stack register levels. The register S4 is at the top of the stack in the document filing analogy. Hewlett-Packard [**?**] referred to it as the bottom of the stack. From now on I refer to this as the **input** to the stack in order to avoid confusion.

In the following example it is convenient to introduce an additional register that we refer to as the **CHARACTER HOLDING register (CHR)**. Though the CHR has no role in RP per se, it has several practical purposes here. One is to provide a register where final output from the serial input can be temporarily stored. Serial streams only provide one digit at a time stored as an ASCII character. Operands will therefore have to be built from digits prior to arithmetic processing. Another application of the CHR is that it makes it easier to implement the RP algorithm.

Exactly how you handle input from the serial input for RP processing is one of the design decisions you will have to make in your project. You do not have to use the CHR approach.

Input:	CHR	S4	S3	S2	S1
4	4	.	.	.	.
ENTER	4	4	.	.	.
2	2	4	.	.	.
ENTER	2	2	4	.	.
5	5	2	4	.	.
*	10	4	.	.	.
+	14	.	.	.	.
ENTER	14	14	.	.	.
1	1	14	.	.	.
ENTER	1	1	14	.	.
2	2	1	14	.	.
ENTER	2	2	1	14.	.
3	3	2	1	14.	.
*	6	1	14	.	.
+	7	14	.	.	.
/	2	.	.	.	.

The effect of ENTER is to push numbers onto the stack while leaving the current digit in the CHR. Note that in RPN operators are never stored on the stack. In the algorithm described here the effect of an operator is that the RP controller pops the stack, triggers the operation and places the result in the CHR.

In this implementation of RP, the ENTER key has to be pressed whenever there is further input after an operator so that the last result is stored on the stack and not overwritten by new input to the CHR. As we shall see, this choice of implementation is by no means unique: the HP-35 handles the storage of prior results in a different manner.

Example 2

(-4 + 54) / (1 + 3 × (7+1))

In RPN this is described by,

4 CHS ENTER 54 PLUS ENTER 1 ENTER 3 ENTER 7 ENTER 1 + × + /

Input:	CHR	S4	S3	S2	S1
4	4	.	.	.	.
CHS	-4	.	.	.	.
ENTER	-4	-4	.	.	.
54	54	-4	.	.	.
+	50	.	.	.	.
ENTER	50	50	.	.	.
1	1	50	.	.	.
ENTER	1	1	50	.	.
3	3	1	50	.	.
ENTER	3	3	1	50	.
7	7	3	1	50	.
ENTER	7	7	3	1	50
1	1	7	3	1	50
+	8	3	1	50	.
*	24	1	50	.	.
+	25	50	.	.	.
/	2	.	.	.	.

It should be clear from this example that in any RP calculation you will never need to access variables lower than the top level of the stack.

2.4 The HP-35 Reversal Polish Algorithm

The following discussion follows articles from the Hewlett-Packard journal describing the HP-35 calculator ([?], [?]). Fig. 3 shows the instructions sticker posted on the back of the calculator and Fig. 4 shows the HP-35 implementation of the RP algorithm.

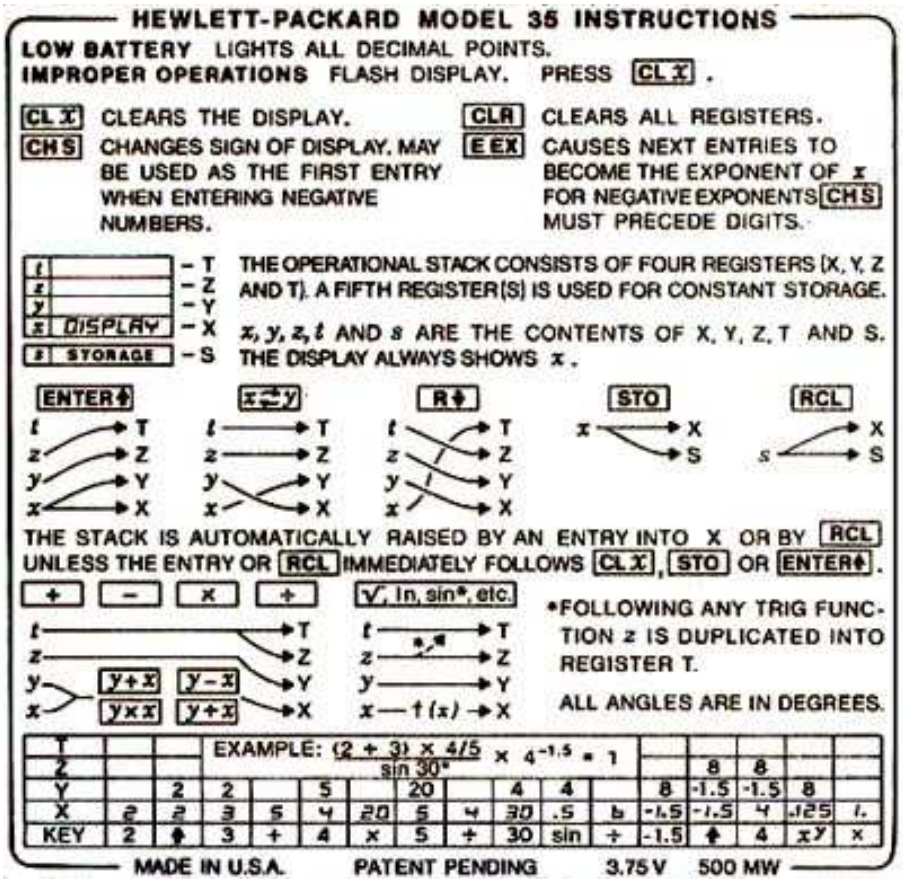
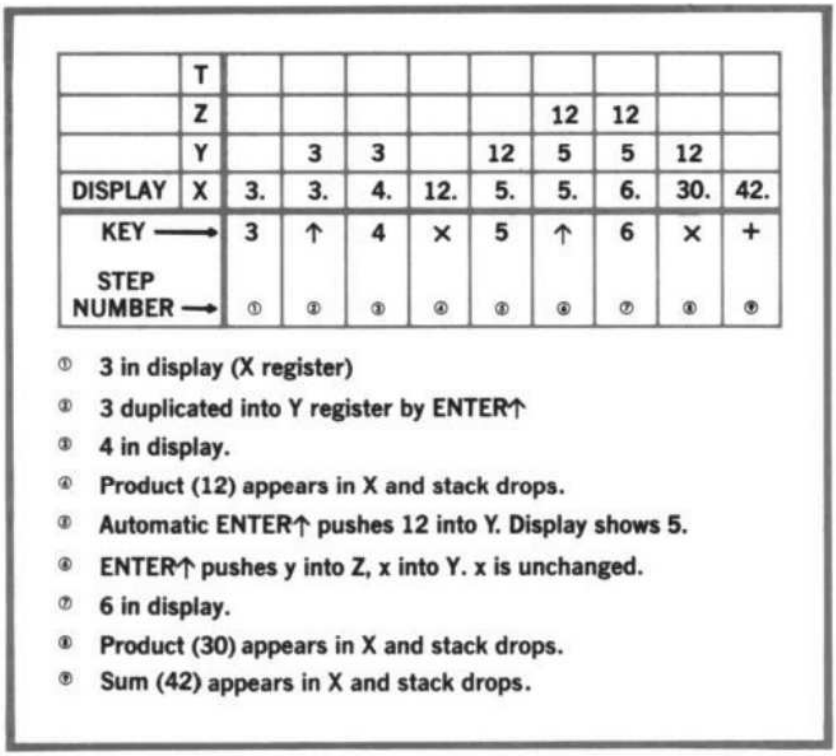

HEWLETT-PACKARD MODEL 35 INSTRUCTIONS
LOW BATTERY LIGHTS ALL DECIMAL POINTS.
IMPROPER OPERATIONS FLASH DISPLAY. PRESS CL x .
CL x CLEARS THE DISPLAY.
CLR CLEARS ALL REGISTERS.
CHS CHANGES SIGN OF DISPLAY. MAY BE USED AS THE FIRST ENTRY WHEN ENTERING NEGATIVE NUMBERS.
EEX CAUSES NEXT ENTRIES TO BECOME THE EXPONENT OF x FOR NEGATIVE EXPONENTS CHS MUST PRECEDE DIGITS.
THE OPERATIONAL STACK CONSISTS OF FOUR REGISTERS (X, Y, Z AND T). A FIFTH REGISTER (S) IS USED FOR CONSTANT STORAGE.
DISPLAY
STORAGE
x, y, z, t AND s ARE THE CONTENTS OF X, Y, Z, T AND S. THE DISPLAY ALWAYS SHOWS x.
ENTER↑
x⇄y
R↓
STO
RCL
THE STACK IS AUTOMATICALLY RAISED BY AN ENTRY INTO X OR BY RCL UNLESS THE ENTRY OR RCL IMMEDIATELY FOLLOWS CL x, STO OR ENTER↑.
√, ln, sin*, etc.
*FOLLOWING ANY TRIG FUNCTION z IS DUPLICATED INTO REGISTER T.
ALL ANGLES ARE IN DEGREES.
EXAMPLE: (2 + 3) x 4/5 sin 30° x 4^-1.5 = 1
KEY
MADE IN U.S.A.
PATENT PENDING
3.75 V
500 MW

Figure 3: The HP 35 instruction sticker.

As you can see the HP implementation differs in a couple of ways from the version presented above. Firstly the CHR in a HP-35 is actually the input register to the stack, X (see Fig. 4) (the display being connected to this register). Secondly, after an operation is executed, results are pushed onto the stack without the need for an ENTER key. Actually the HP-35 algorithm does allow the user to press the ENTER key after an operator. However this has exactly the same effect as not pressing the ENTER key: so it is probably ignored. The reason for this design decision appears to be to reduce the number of ENTER key strokes used in lengthy calculations. In my experience one of the weaknesses in the engineering of the HP RP calculators was the tendancy of keys to stick after extended use.



	T									
	Z						12	12		
	Y		3	3		12	5	5	12	
DISPLAY	X	3.	3.	4.	12.	5.	5.	6.	30.	42.
KEY →		3	↑	4	×	5	↑	6	×	+
STEP NUMBER →		①	②	③	④	⑤	⑥	⑦	⑧	⑨

① 3 in display (X register)
② 3 duplicated into Y register by ENTER↑
③ 4 in display.
④ Product (12) appears in X and stack drops.
⑤ Automatic ENTER↑ pushes 12 into Y. Display shows 5.
⑥ ENTER↑ pushes y into Z, x into Y. x is unchanged.
⑦ 6 in display.
⑧ Product (30) appears in X and stack drops.
⑨ Sum (42) appears in X and stack drops.

Fig. 1. *HP-35 Pocket Calculator has a four-register operational stack (last-in-first-out memory). Here's how the stack works to solve (3 × 4) + (5 × 6). Answers appear in display register X, in floating-point or scientific notation, to 10 significant digits.*

Figure 4: The HP-35 RP implementation explained.

3 RPC Hardware Description

3.1 General

The overall block diagram of the RPC is shown in Figure 5.

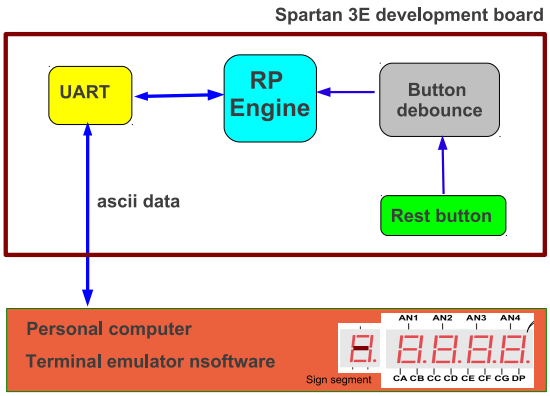


Figure 5: Reverse polish calculator block diagram

The system in the red square is implemented in the FPGA. The orange block is a personal computer with a serial interface. The subsystems inside the FPGA include the **reverse polish calculator (RP engine)** that does arithmetic calculations in reverse polish, the **UART (Universal Asynchronous Receiver Transmitter)** block that handles serial communications with the PC and a **switch debouncer** for the FPGA RESET push-button. A UART consists of a serial transmitter and a serial receiver. The UART is a separate design project from that of the RP engine and will form part of the phase one project milestone.

In the rest of this chapter the detailed characteristics of the design are described.

3.1.1 Overview

The project may be summarised as follows.

1. The **PC** uses its serial port to transmit **ASCII** characters representing RP commands to the RP engine. The PC also receives the output of the RPC in serial.

2. The RP engine receives these characters via the **Universal Asynchronous Receiver Transmitter (UART)** block inside the FPGA. This is simply a matter of entering reverse polish commands from a computer keyboard to a terminal emulator and sending these as ASCII characters over a serial line. These commands will

be echoed inside a terminal emulator responsible for controlling the serial port. A **terminal emulator** is a software application that transfers characters over a serial connection to a peer and displays the characters received. Optionally the transmitted characters can also be locally echoed which is preferable in this application.

3. **Universal Asynchronous Receiver Transmitter (UART)** core inside the FPGA communicates in serial with the PC using the terminal emulator on the PC. The UART receiver in the FPGA receives characters from the PC. The RP engine sends results to the PC via the UART transmitter. You will need to study the electronic aspects of serial communications in detail. Below we define the precise parameters for the serial port.

4. **The RP engine** may either be an **RTL** based design or a straight **datapath** consisting of digital blocks to perform arithmetic calculations. These blocks usually include an **Arithmetic Logic Unit (ALU)**, a **stack** and various supporting combinational and sequential components. If you study the VERILOG code for the MU0 controller ($mu0\_ctrl.v$) you will see that processing decisions are based on the **opcodes** of the command executed. For the RPC they are the RPC command ASCII characters sent from the host PC. **This design forms the second milestone for the project and is by far the most creative and difficult part of the project. You should start to study Reverse Polish notation immediately. There are various levels of the design that lead to an ALU that is quite complex.** .

3.2 Spartan 3E Board Peripherals

Fig. 6 shows the Spartan 3E Starter board peripherals. The DCE (Data Communications Equipment) port is used for communications with the PC. The DCE connector is a female DB9 connector as would be found on a modem. Push button SOUTH will be used to reset the RP engine to an initial state. It can be also used to resolve hardware freezes if they occur. In fact the reset button is a part of the power up phase of any digital device designed to leave the calculator in a *ready* state. *Reset buttons are therefore advisable in all designs.*

3.3 Switch Debouncing

The use of push-buttons in a digital application requires switch debouncing. During normal operation, a mechanical switch bounces on its contacts leading to multiple makes and breaks. Bounces may last for several milliseconds and could confuse digital devices connected to the switch. Switch debouncing is a technique that eliminates this problem by presenting an input to the design only once the bounces have settled. Detailed information sources on debouncing have been posted on WATTLE under *Documentation and Reading material*.

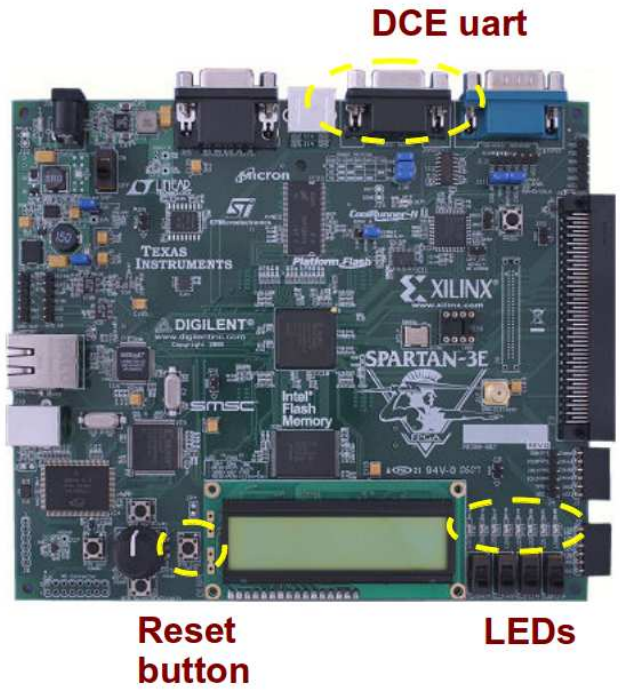


Figure 6: S3E board showing the peripherals.

A switch debouncer design is shown in Fig. 7. The circuit consists of a chain of three D-type flip-flops and a binary counter like that used to divide the system clock in hlabs. The tap connected to the counter is chosen for best performance and usually by trial and error. Depending on which counter bit is tapped, switch bounce glitches may or may not pass through to influence the output of the AND gate and hence be registered as a change. To obtain all 1's at the input to the AND gate, the switch voltage must be held high for three cycles of the divided *sysclk*.

A switch debouncer along these lines will provided to you. You should try to implement the switch debouncer in a separate hardware design to make sure that it works. Use push button SOUTH and send the output of the bouncer to one of the S3E LEDS. Design it so that each time the button is pressed, the LED is toggled. Check that the LED toggles reliably no matter how you press the button. If the odd bounce leaks through then sometimes the LED will not toggle. You will need to fiddle with the counter tap until it works for the particular push button. It is important that the debouncer work reliably before you use it as a component in a complex project.

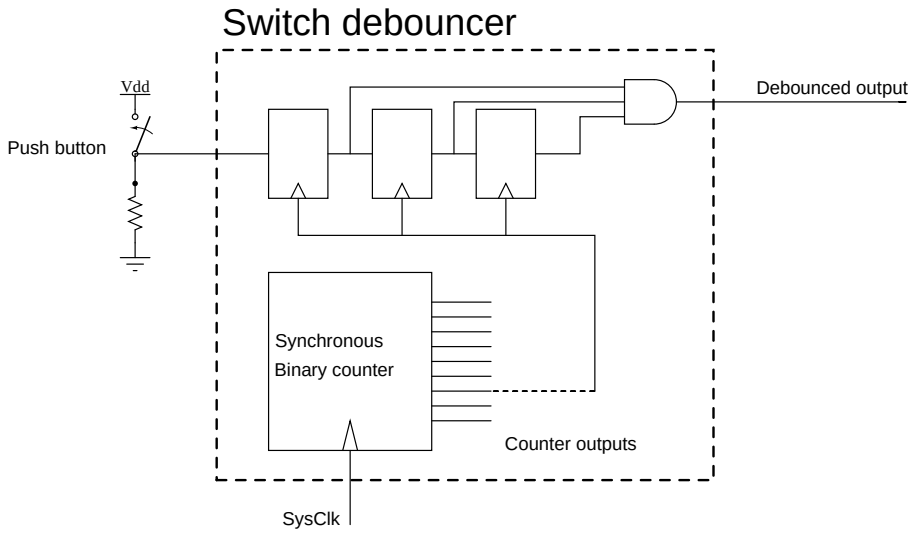


Figure 7: Push button debouncer.

3.3.1 Basic Character encodings and Communications

Characters will be sent from the PC to the FPGA and vice versa using ASCII code. ASCII stands for the American Standard Code for Information Interchange. Its purpose is to provide a way of representing letters of the alphabet, punctuation characters and numbers in digital format. It is the standard method for sending alphanumeric data between digital systems. Since all letters of the alphabet (lower and upper case), the digits and other symbols number aroung ninety, a 7 bit standard code was proposed. Since a 7 bit code can only represent 128 characters however it is rather restrictive. For foreign language support for example, the **UNICODE** set has been devised. ASCII is a subset of UNICODE. The table of ASCII character representations can be found widely on the WEB so take a look for example at `http://www.asciitable.com`, https://en.wikipedia.org/wiki/ASCII.

In all levels of the calculator (see below), the calculator is to produce four signficant decimal figures of precision with an additional potential (minus) sign and decimal point. All number entry and output display is to be in decimal. Entering numbers in excess of four digit precision should be allowable but results must be presented with four significant figures.

In addition to knowing the encoding scheme, we also need to discuss the protocol by which the calulator exchanges data with the UART. When data arrives over the serial line from the PC it does so *asynchronously*. The UART decodes these characters into bytes that are temporarily stored. In a PC UART there is a receive buffer register (RBR) capable of storing 14 bytes temporarily. When this buffer contains data, an interrupt is sent to the microprocessor on your PC to alert it to the presence of data. This interrupt is one of the IRQ's available to the INTEL processor. Likewise there is a transmit buffer register (XBR) where data ready for serial transmission is stored by the processor. The reason for

the registers should be clear. The disparity in the speed of serial communications (9600 baud means 9.6 kbps!) and that of the CPU clock means that data sent between the CPU and the serial port UART needs to buffered or it could get lost. Even though the CPU is much faster than the serial port, the RBR is still needed because the CPU is busy handling all tasks in the operating system (OS).

In the RP calculator too there is a disparity between the 9600 baud of the UART and the 50MHz system clock of the FPGA. There are said to be two **time domains** in the design. There is more than one way to allow the data to *cross the time domains*. One way is to use an **interrupt** to enable entry of the latest character into a memory in a manner similar to the RBR in a PC that avoids **buffer overruns on the serial port**. Employing a **FIFO** at the output of the UART is a good approach. All you need do is make sure that the time taken to do a calculation (on the order of 100's of ns) is short enough that the FIFO does not overflow at the BAUD rate. The FIFO is read synchronously by the system clock. To include a FIFO you may either design your own in Verilog using the examples on the course website or you may include one of the **Xilinx ip-cores** in the design.

Another solution to the time domain problem that avoids the need for both an interrupt and a FIFO is to instanteously enter the characters into a stream of either character codes or a special unique non-character code at every cycle of the system clock. The CHAR/NOCHAR system is described in Fig. 8. The RP engine receives and interprets the CHAR characters that contain the RPC commands and data, while ignoring the NOCHARS.

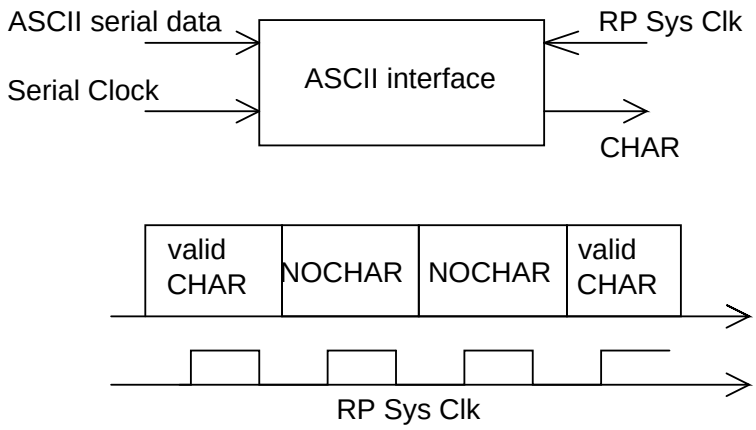


Figure 8: The timing diagram shows how the serial output characters (CHARs) on the posedge of the RP system clock are read and how the NOCHAR character is produced

3.3.2 Serial Communications

In order to display output from the calculator and to communicate with the calculator from a PC, we will use the DCE (Data Communications Port) on the S3E board as shown in Fig. 6. By connecting the DCE port to a DTE (Data Terminal Equipment) interface by a straight through serial cable we will be able to communicate with the calculator using a terminal emulator application on the PC.

Serial communications will use 8N1 (8 data bits, one stop bit and no parity check bit) at 9600 baud with no flow control. These parameters will have to be configured in the terminal emulator before communications can occur.

3.3.3 Character Display in the Terminal Emulator

Display of characters in the terminal emulator must include both the characters sent (local echo) as they are on a HP-35 and the result received from the FPGA. On a HP-35 there is only a single line of thirteen 7-segment LEDs. There is no possibility of carriage return as there is with a terminal emulator. Numbers are displayed digit by digit. When a *CHS* is pressed, control jumps back to insert the minus sign. Every time the *ENTER* key is pressed, the display is cleared and a new number is entered. When an operator is pressed the display is cleared and the result (even if temporary) is displayed. We can reproduce this behaviour in a terminal emulator. Using ASCII to encode the data and display it is one key part of the solution. To alter the way the characters are displayed in the terminal emulator, we will write a C-program to (a) take the characters entered from the keyboard and display these locally in the xterm and (b) to transfer the RPC commands and data over the serial link to the FPGA. Sample programs that demonstrate how to do this are provided.

The terminal emulator provided is called *cterm.c*. It runs inside a LINUX xterm. We can use the Ubuntu Virtual Machine to compile and run *cterm* and to send characters to the UART core in the FPGA. The VMplayer application that runs the Ubuntu VM can be configured to communicate with the hardware serial port or a USB-serial adaptor on the host. Note that *cterm* works well enough to establish communications with a UART core in the FPGA and therefore to serve the purposes of the first milestone for the project. However in order to make it a sensible interface for an RPC like the HP-35 requires some minor adjustments of the exact terminal emulation are required. To do this a couple of other programs may prove useful. The program *ascii_print.c* demonstrates how ASCII characters can be created from keyboard input using C. The program *rpc.c* is a simplified Reverse Polish Calculator in C that demonstrates some aspects of terminal emulation that crudely mimicks the HP-35. This design goal is set largely for aesthetics and to avoid certain bizarre alternatives.

It is important to note that the UART core will receive a certain standard set of ASCII characters that represent the RP commands and data and will

return results in a similarly well defined format. The issue of terminal emulation is purely one of presentation of the results to the user. Thus the exact presentation in the terminal to the user is not of primary importance in the project.

3.3.4 Possible Architectures for the RPN Engine

It is most natural to design the calculator as an RTL system just like MU0. You may even consider using MU0 itself and adapting it by the introduction of new **opcodes** to specialise in RPN processing. However there are an infinite number of ways to build a satisfactory calculator: even a pure datapath that has no control at all and is hardwired to handle RPN processing. You may even use the approach implemented by Hewlett-Packard in their original design of the HP-35.

How you process data in the RP engine affects the conversions from ASCII that you will need to make in order to produce the binary numbers on which you can do binary arithmetic. For example, you may perform mathematical operations in two's complement binary or perhaps BCD (binary coded decimal). If you use the former then arithmetic is straight forward but you will need to convert the binary digits (bits) to ASCII format. If you use BCD then arithmetic will be less obvious. Justifying your design decisions for the **ALU** is necessary for the mid semester report.

Given the implementation of the RP algorithm described above and following the RTL description of MU0, one may propose the RTL architecture shown in Fig. 9.

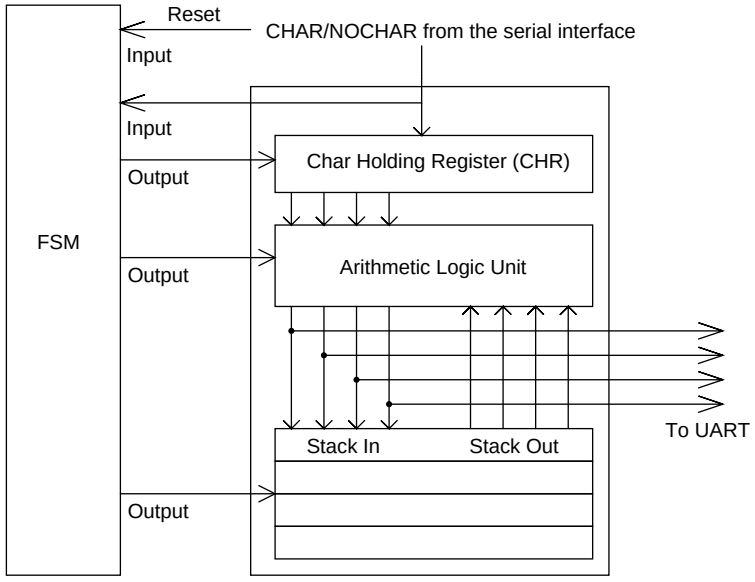


Figure 9: Simplified RPN RTL control and data paths.

In this figure, the control path is a FSM (at the left) which has two inputs: the CHAR/NOCHAR code and a reset. These are synchronised to the system clock. From the above description of the CHR, exactly one CHAR/NOCHAR appears per positive clock edge for each valid CHAR. Otherwise a NOCHAR is produced. The reset must be provided via a separate push button input and not from the keyboard so that the calculator can be manually forced into the INIT state (commonly referred to as "switching on the calculator").

The CHAR/ NOCHAR inputs are analogous to the **opcodes** stored in memory in MU0. These determine the state transitions of the RP controller FSM. The outputs of the controller are a bunch of enable and reset switches that control the hardware blocks of the datapath. As is the case for MU0, **there should no need to send the data buses through the controller FSM (see Fig. 9.) Notice that the datapath consists of hardware blocks that should be already very familiar to you**. In the present example these are the **Char holding register**, **an arithmetic logic unit** and a **stack**.

Fig. 10 shows in basic form a pure datapath version of the calculator.

3.3.5 Important Advice

The calculator is a complex project with many building blocks and many new concepts. To tackle such complex designs you should proceed as follows:

1. **All sequential designs must be system clock synchronous.**
2. **Always draw a block schematic with component inter-connects accompanied by detailed timing diagrams that show where you expect transitions to occur with respect to the system clock before you consider translation into Verilog HDL. An excellent example of how to do this for the serial echo implementation in part 1 of the project is provided in the next section.**
3. **Always thoroughly understand the operation of your design and simulate each module and the entire system (if possible) in Icarus Verilog and GTKwave before attempting an ISE WebPACK design flow. Implementation in hardware should only be attempted after you are convinced that all Verilog syntax is correct and consistent with what I teach in the course and the GTKwave simulations concur with your design goals.**

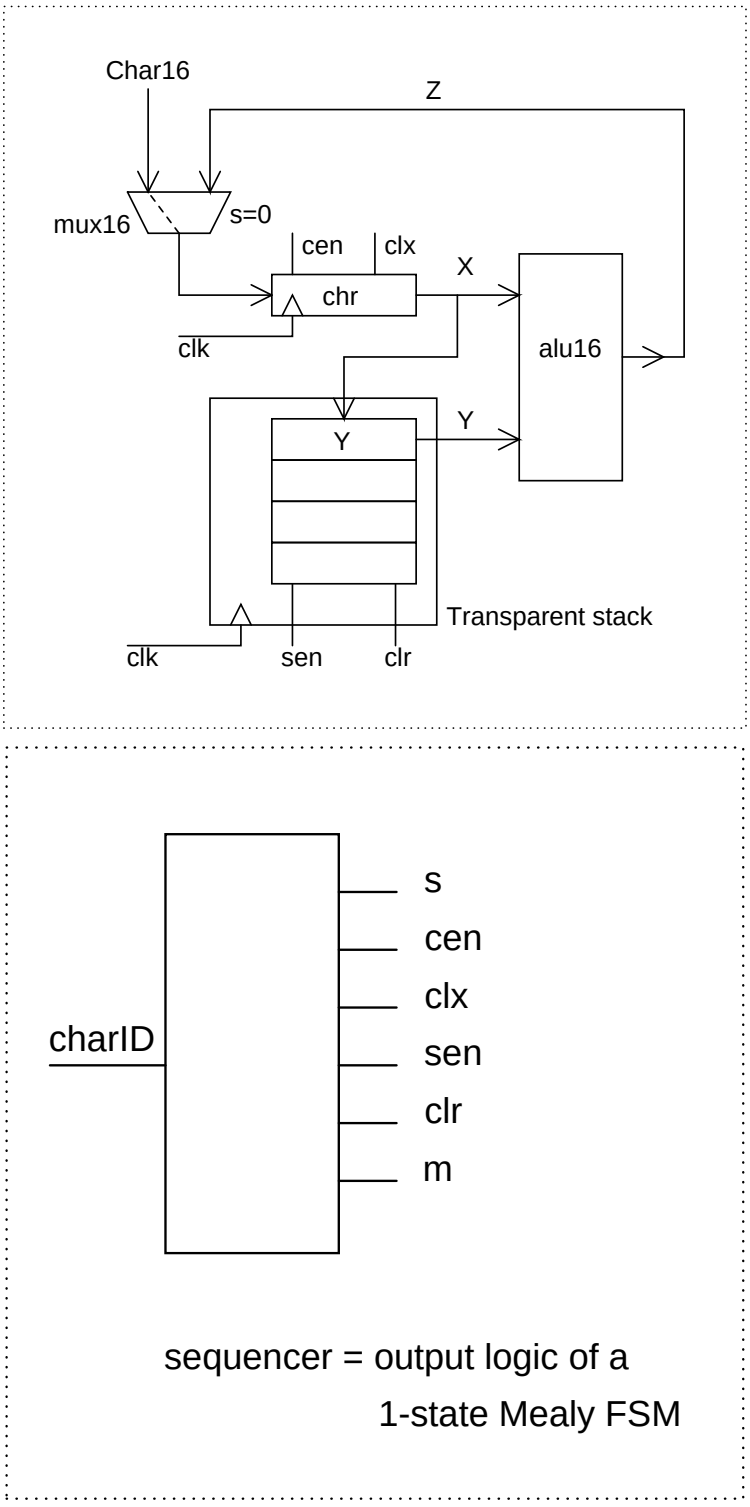


Figure 10: A datapath only RP engine design.

4 Project Requirements

The project will involve the design and implementation of up to three reverse polish calculators complete with a serial communications and a terminal emulator user interface. The only difference between the three calculators lies in the arithmetic capabilties of their respective arithmetic logic units. You may otherwise reuse as much of your code in each design.

4.1 Calculator Reset

You must have a debounced calculator reset. Use push-buttom SOUTH for calculator reset.

4.2 Implementation Levels of the RP Engine

In the following sections different levels or versions of the RP engine are described. The levels correspond to increasingly complex implementations and improvements in functionality of the calculator. The changes only affect the design of the arithmetic logic unit. Thus you can reuse your UART etc. Levels 1 and 2 are compulsory. Level 3 is not compulsory but attracts a 5% mark bonus.

Make sure that you implement either the direct or the HP Reverse Polish algorithm previously described. You must STRICTLY ADHERE to the proposed ASCII conventions (see the section on assessment).

These levels are

1. A signed four digit decimal integer calculator that does addition and subtraction.
2. A signed fixed point four digit decimal calculator that does addition, subtraction, multiplication and division. There are two significant digits before the point and two after.
3. A floating point signed four digit decimal calculator that does addition, subtraction, multiplication and division.

In order to obtain full marks in the project it will be necessary to complete levels 1 and 2. Level 3 attracts extra marks on top of the course 100% total. Much of this requires a good understanding of number systems and number representations. Those who have not done the COMP2300 course may find the following lecture notes useful.

`http://cs.anu.edu.au/student/comp2300-2009/lectures/`

4.2.1 RP Engine Level 1

We confine ourselves to decimal integer addition and subtraction. Key functions will be entered into the terminal emulator. To illustrate the functionality at this level consider Figure 11 showing the front panel of a HP-35 calculator. The relevant keys are shown inside the yellow square.

Figure 11: HP-35 functionality for the level 1 system.

The large blue key on the top left is the ENTER key. The operator keys − and + are in blue at the left. The **CHS** button changes the sign of the current number on the display and **CLX** clears the display to a 0. The **CLR** key clears the stack. At this level we will not implement the keys that have a red cross through them. These include, among many others, the **EEX** key which converts a number to scientific notation, the **PI** key which stores the number π and we will not need the decimal point key.

The following table shows the keyboard characters and their ASCII values we will use to represent the HP-35 function keys of Figure 11. The same ASCII characters must be used for the output from the calculator as well. You should confirm the ASCII values by looking up the tables on-line or running the program *ascii_print.c*. Note the the numeric pad on the keyboard must produce the same ASCII encodings, so it does not matter whether you use the QWERTY part of the keyboard or the numeric pad to enter numbers and operators.

RP Function	Keyboard key	ASCII (3 digit octal)	Description
$ENTER$	"e"	145	Store on stack
$-$	"$-$"	055	Subtract
$+$	"$+$"	053	Add
CHS	"s"	163	Change sign of last number entered
CLX	"x"	170	Clear the display to 0.
CLR	"r"	162	Clear all stack levels to 0.

At this level that you develop your basic RTL design. The arithmetic is not so hard and the ALU is just a place holder. This is the most important project milestone. Try to make it extensible to the more complex ALU designs. The precision is to be the full 4.0**: that is four decimal digits with no decimal point.**

The HP stack has four levels. The stack in the project (not including any Character Holding Register (if you choose to have one)) should be at least four levels deep.

Handle overflows by displaying a row of 4 asterisks in the seven segment displays.[1]

4.2.2 RP Engine Level 2

The aim here is to reuse the RTL design of level 1. The only difference is to implement fixed point arithmetic with fractional decimals. Fig. 12 shows the HP-35 keys.

At this level we include multiplication, division and a fixed decimal point in the middle of the display. The precision is 2.2 **meaning two digits before the decimal point and two after.**

The following table shows the meaning and keyboard designations of the HP-35 keys of Figure 12.

[1]Interestingly the HP-35 fails to handle overflow properly. The HP-35 rounds overflowing results down to the maximum number $9.999999999 \times 10^{99}$. Dividing any two numbers larger than this by each other produces a 1.

Figure 12: HP-35 functionality for the level 2 and 3 systems.

RP Function	Keyboard key	ASCII (3 digit octal)	Description
$ENTER$	$''e''$	145	Store on stack
$-$	$''-''$	055	Subtract
$+$	$''+''$	053	Add
$\times$	$''*''$	052	Multiply
$/$	$''/''$	057	Divide
.	$``.''$	056	Decimal point
CHS	$``s''$	163	Change sign of last number entered
CLX	$``x''$	170	Clear the display to 0.
CLR	$``r''$	162	Clear all stack levels to 0.

4.2.3 RP Engine Level 3

In this optional case we aim to implement floating point arithmetic (without scientific notation i.e. ignore the **EEX** key). The floating decimal point in the result adjusts itself to the appropriate position on the display to maximise precision. Floating point will allow us to multiply decimal numbers with larger dynamic range than fixed point.

The precision is 4 digits maximum before the decimal point and 3 digits maximum after the decimal point with a digit before.

Since we will not implement exponents, the HP-35 functionality is the same as in Fig. 12 followed by the same table above showing the ASCII designations.

4.3 Assessment

Assessment involves the following items. The project will be worth 40/100 marks in total for the course (45/105 for those who attempt level 3). The first part of the project is comprised of a short design report (eight pages) and a working serial echo-UART core implementation due by C.O.B on the 23rd of April just after the mid-semester break (20%). The second part of the project consists of the implementation of levels 1,2 and

perhaps 3 provided in Verilog HDL form only: schematics, verilog source modules, UCF file and NGC files pertaining to any ip-cores you use. Do not provide bit-files. There will be hardware tests awarding marks for successful implementations. Part 2 is due at the end of semester and is worth $20\% - 25\%$. All assessment items should be uploaded to WATTLE. Do not provide hard copies.

4.3.1 The Mid-Semester Design Report

In this exercise you are to concentrate on the digital system of Fig. 9 or Fig. 10 or whatever you choose to do. Provide the following.

1. A short introduction describing in no more than half a page the approach taken. What were the key aspects of your implementation of the algorithm and how it impinged upon the architecture. **2 marks**

2. A description and drawing and timing diagram of all hardware blocks that you think make up the calculator,

   (a) RTL approach: FSM (instead of a schematic use next state tables and/or state diagrams as appropriate, Karnaugh maps if you think appropriate), ALU (if combinational provide the schematic and the truth table of a 1-bit fulladder or whatever), stack, CHR and any other basic block.

   (b) Datapath approach: Sequencer, ALU, stack, CHR, etc. You may follow the example Fig. 10.

   (c) MU0 assembly language approach: an algorithm based on assembly language commands including any new ones you may propose (you may follow *rpc.c*). Describe the new hardware blocks of MU0 that you need.

   A hardware block is a schematic that includes all inputs and outputs. In order to make the description precise, for sequential blocks (other than FSMs) also provide a timing diagram that shows the relationship of all input and output signals to the system clock. A good example is Q2 in the midterm exam. For a combinational block (a timing diagram means nothing) - instead provide the truth table. These descriptions should be sufficiently precise as to allow you to write the Verilog for synthesis directly. You do not need to provide schematics consisting solely of gates as this would mean that you had specified the rtl Verilog implementation directly thus obviating the advantages of the behavioural approach. **6 marks**

3. For the level 1 system provide test benches and GTKwave simulation traces demonstrating individual working hardware blocks. Do not attempt an ISE design flow or an simulation for the RPC calculator in toto. There is no need to provide any test benches for the complete operating calculator - just the individual blocks in the RTL system. These may be located in an appendix thus exceeding the eight page limit. **3 marks**

4. A short description of how you would implement the arithmetic system blocks for multiplication and division in the level 2 ALU. **2 marks**

5. Overall VERILOG coding style (to be marked from your sources) - **2 marks**

6. The Verilog HDL implementation of the uart complete with test benches, top module and switch debouncer. Provide source Verilog, UCF and NGC files (if any for ip cores such as the fifo) only - no ISE project or bit-files. Thus the uart is to be provided ready for implementation in hardware. **5 marks**

7. **The mid-semester project report is worth** 20% **of the final mark and must be handed in by C.O.B. the 23***rd* **of April.**

8. Upload to WATTLE the report in PDF format. Use the naming convention for the report UXXXXXX-NAME-RPC-REPORT-ENGN3213-2012.pdf inside a ZIP-file named UXXXXXX-NAME-RPC-REPORT-ENGN3213-2012.zip containing your UART code.

The report should total no more than eight pages not including appendices.

4.3.2 The Mid-Semester Implementation of the Serial Echo Terminal

In the first part of the project the UART is implemented in FPGA. The aim is to build a serial echo core based on the UART that receives data in the FPGA from the terminal emulator and echoes these back to the terminal emulator. This simple system will prove that the UART is working as required prior to implementation in the RPC.

The overall picture of serial communications is shown in Fig. **??**.

In this section we describe the hardware implementation of the complete serial echo core. The instantiation templates, schematics and timing diagrams presented here provide an excellent example of how you should go about your designs.

The UART instantiation template is as follows.

```
module uart(
       input wire sysclk,         //system clock in
       input wire reset,          //push button hard reset in from debouncer
       input wire Sin,            //serial in from serial line
       output reg [7:0] rx_byte,  //receive data byte out after deserialisation
       output reg rx_rdy,         //receive data ready now valid

       output reg Sout,           //serial out to serial line
       input  wire [7:0] tx_byte, //transmit data byte input to be serialised
       input  wire tx_rdy         //transmit data byte input valid
)
```

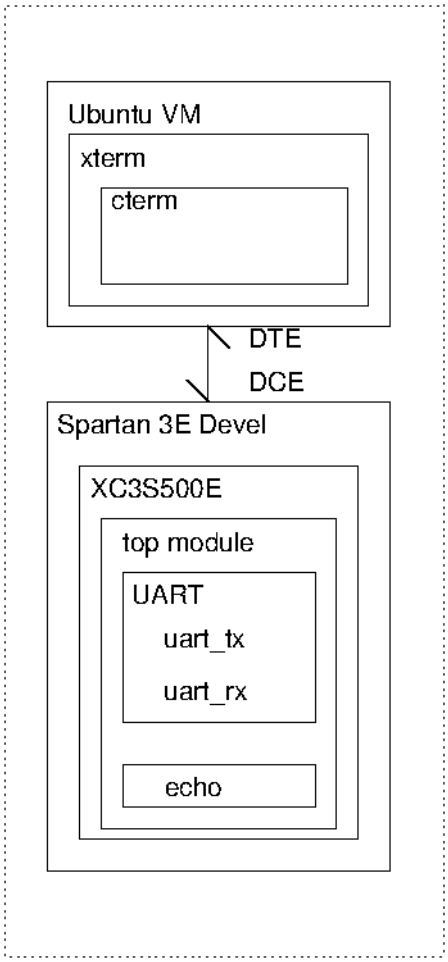


Figure 13: Serial communications system showing where all the components are located. The terminal emulator *cterm* runs in an *xterm* application on the Linux Ubuntu Virtual Machine. The uart and the echo always block are hardware blocks running on the FPGA

Note the use of the mandatory hard reset. This reset is used to reset the uart to a known state. For example if your design involves a FSM, then the reset may be used to reset the FSM to the *'init* state. The use of push button SOUTH will also require you to implement the switch debouncer. Code for the switch debouncer will be provided on WATTLE for you to include in your project. You will also need to provide a UCF file.

The serial echo core should provide a working implementation of your UART. The only difference is that a line in the top module (the one containing uart as a test module) has to be included to allow echoing. In Verilog HDL the top module would look as follows:

```
module uart_top(input wire system_clock,
                input wire push_button,
```

```
                input wire serial_bits_in,
                output wire serial_bits_out,

//Switch debouncer
swdebouncer swd(system_clock, push_button, push_button_debounced);

uart unice(.sysclk(system_clock),
       .reset(push_button_debounced),
       .Sin(serial_bits_in),
       .rx_byte(received_serial_byte),
       .rx_rdy(receive_ready),
       .Sout(serial_bits_out),
       .tx_byte(transmit_serial_byte),
       .tx_rdy(transmit_ready)
);

//Echo always block ...
always @(posedge system_clock) begin
    if(receive_ready) begin
           transmit_serial_byte <= receive_serial_byte;
           transmit_ready <= 1'b1;
    end else begin
           transmit_serial_byte <= 8'h00;
           transmit_ready <= 1'b0;
    end
end
```

Fig. 14 shows the architecture to be implemented. Pay attention to the detail in this figure and try to understand what is happening. Also try to understand how the hardware blocks relate to the Verilog above. The timing diagram is drawn with respect to the baud clock of 9600 bps. This clock is produced within the UART by a suitable clock divider (see lecture 4 for an idea about how to make such a divider using a counter and a decoder). Of particular importance is exactly how the timing operates in the echo always block at the 50 MHz sysclk rate. This is shown in detail in Fig. 15. Finally note that serial communications is **little endian***: the least significant bit is sent first: hence the result* $8'hb9$. Fig. 16 shows the details of the RS232 frame structure.

4.3.3 UCF File

In your projects you must use the following UCF file.

```
NET "sysclk" LOC = "C9";
```

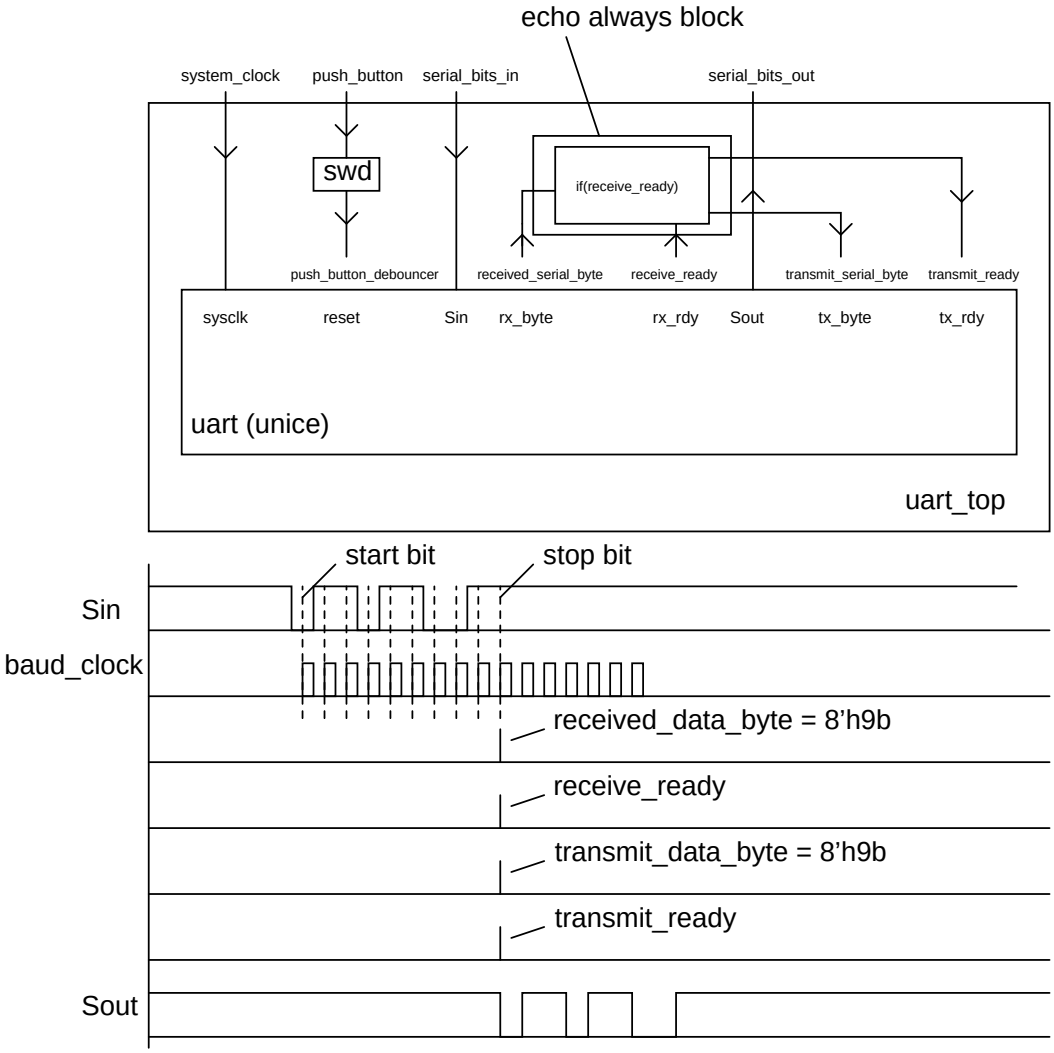


Figure 14: Serial echo core showing baud rate timing.

```
NET "push_button" LOC = "K17" | IOSTANDARD = LVTTL | PULLDOWN ;
NET "send" LOC = "H13" | IOSTANDARD = LVTTL | PULLDOWN ;
NET "serial_bits_in" LOC = "R7"  | IOSTANDARD = LVTTL ;
NET "serial_bits_out" LOC = "M14" | IOSTANDARD = LVTTL | DRIVE = 8 | SLEW = SLOW;
```

4.3.4 The End-of-Semester Hardware Tests

You provide three calculators at each level that uses PC serial communications. For these tests you provide:

1. A short manual (see below) (**5 marks**)

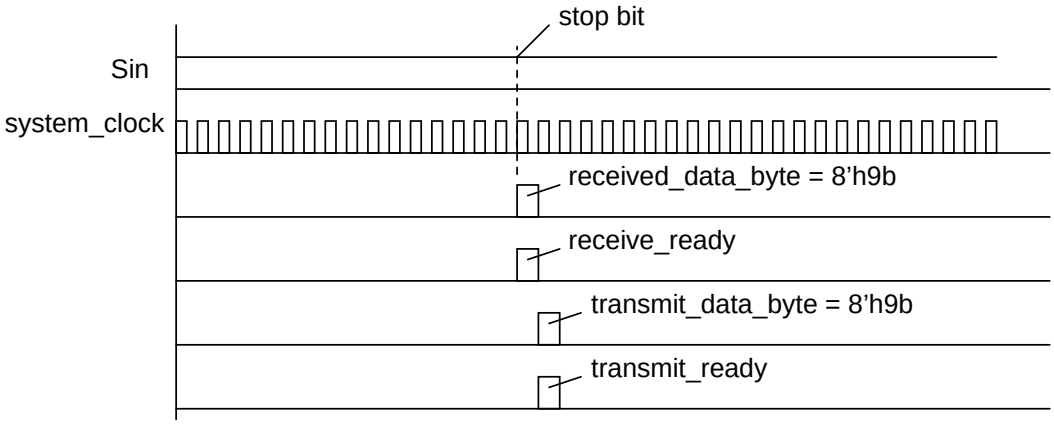


Figure 15: Echo always block sysclk timing

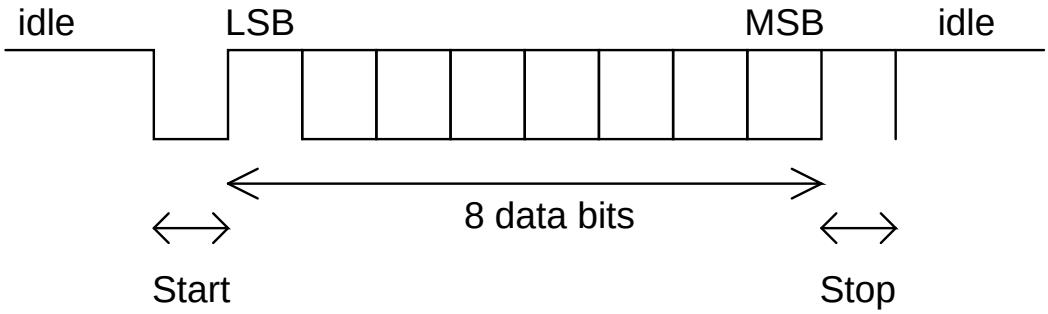


Figure 16: Frame structure of an 8N1 serial frame. The LSB is the least signficant bit and MSB is the most significant bit

2. Hand in separate source code and UCF files in separate zipped folders each labeled according to levels 1,2 and 3 ready for design flow implementation. Do not include bit files (**5 marks** for coding style)

3. Hardware tests involving an arithmetic examination of the calculator (**10 marks for complete and correct levels 1 and 2**)

4. **The manual and three levels are to be uploaded to WATTLE by C.O.B. the 4th of June.**

Provide test benches and compile scripts for each main module ready to go for siumulation in GTKwave (for example FSM, ALU, stack) in each of your three designs but do not provide the GTKwave simulation traces themselves). Compress these, the manual and the three level zips separately into one ZIP file and upload to WATTLE as one file with naming convention UXXXXXX-NAME-RPC-CODE-ENGN3213-2012.zip or UXXXXXX-NAME-RPC-CODE-ENGN6213-2012.zip.

I will take the code that you upload to WATTLE and build this in ISE on the day.

For this assessment item to work your calculator must send and receive the correct ASCII sequences to and from the FPGA. If you cannot get this to work make sure you obtain assistance. This will already be a problem in the first part of the project. Students will not be allowed in the hardware labs during the tests.

4.3.5 The RPC Manual

Provide a soft copy of the user manual totalling about one page that contains a block schematic of the calculator not showing ALU details, instructions if necessary and known issues (if any) for the examiner (me).

4.4 Project Rules

This project leaves plenty of scope for individual creativity. You do not have to follow the exact procedure described above for the RTL design of the calculator. If you do choose to be creative in your coding style then I expect a solid justification in your manual and synthesised hardware.

The following are the project rules.

1. You must work alone.
2. You should observe the design conventions introduced in this document and the Verilog coding style introduced in the course.
3. The length of the mid-term report should be $\approx$ 8 pages. There is no pressure to produce a big report and there will be no penalties for exceeding the limit. You should add your Verilog source into an appendix. The length of this appendix is not counted in the eight pages.
4. For testing purposes we do not require your bit files. We will need a zipped folder containing the VERILOG modules in a suitable form for loading directly into ISE WebPACK 13.3 (as in IR 103). The design flow should execute continuously and free of errors. We should not have to do any PIN assignments. **Consequently a UCF file to make the designs IO-consistent will be provided.**. Please place it in each of your source folders in each of your source folders. Also provide all NGC files of any ip-cores that you use. You must check that the design flow works from scratch in each new project in ISE WePACK 13.3 *BEFORE* you upload your code.
5. **You may not use any third party code. All Verilog code is to be the original work of the student save Xilinx ip-cores and code offered for general use in the course.**

6. 20% off per day for late submissions.

A Description of the MU0 Microprocessor

A.1 Introduction

The definition of the instruction set shown in Fig. 17 and the requirement of two clock cycles for an instruction forms the specification of MU0.

MU0 Instruction Set

F	Mnemonic		Description
0	LDA	S	Acc := [S]
1	STO	S	[S] := Acc
2	ADD	S	Acc := Acc + [S]
3	SUB	S	Acc := Acc – [S]
4	JMP	S	PC := S
5	JGE	S	If Acc >= 0, PC := S
6	JNE	S	If Acc ≠ 0, PC := S
7	STP		Stop

Figure 17: MU0 assembly language instructions

The first thing to do is to understand what goes on with these instructions. Note the syntax of the commands. The symbol S refers to a memory address. The notation [S] refers to the contents of the memory location.

Consider the datapath of Fig. 18. It shows the following hardware systems,

1. A **program counter register** (PC) which stores the address in the memory of the current instruction. Exactly what is the current instruction and what is the next instruction we'll see in a minute. The addresses count from 0 upwards and in any program, the instructions are stored in the first contiguous memory locations while the data is store in the subsequent locations. This is the basis of the Von Neumann architecture wherein program and data are stored sequentially in memory.

2. An **instruction register** (IR) which contains the instruction while it is being executed.

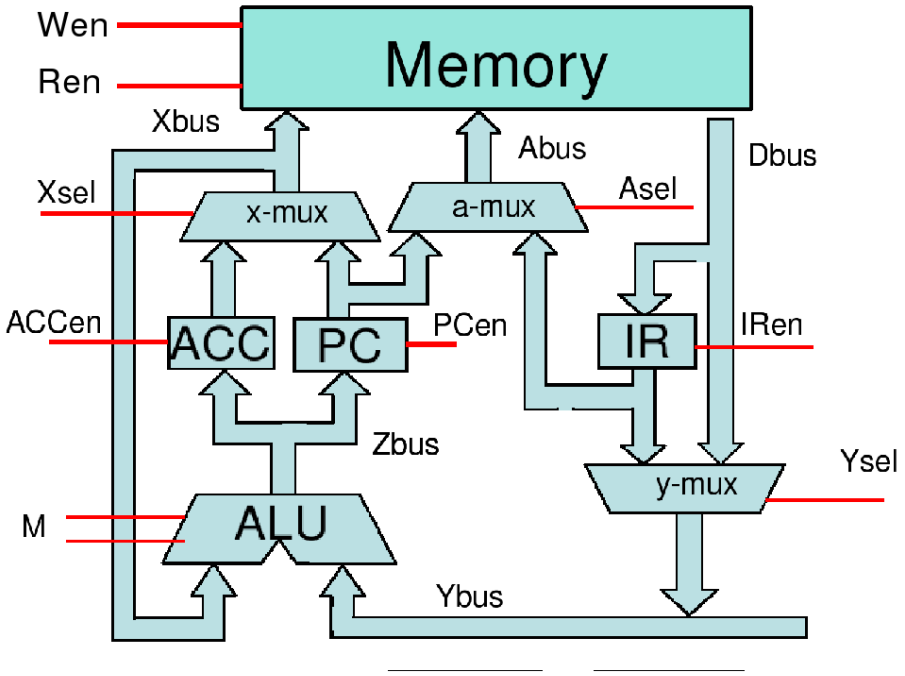


Figure 18: MU0 architecture.

3. An **accumulator** (ACC) which provides intermediate storage of data during instruction execution. The ACC is sometimes referred to as the **working register**.

4. **An Arithmetic Logic Unit** (ALU)

5. Several multiplexers

In MU0 the data has 16 bits and the memory has storage locations that are 16 bits wide. The data in memory is stored at locations that can be located by their address. These addresses are represented by words that are 12 bits wide. That is MU0's memory has 2^{12} memory locations where data can be stored.

It is interesting and entirely pertinent to note that an instruction word consists of 16 bits and can therefore be stored in memory. The most significant four bits ([15:12] in VERILOG parlance) of the instruction word is referred to as the **opcode**. This is the machine language symbol that represents an instruction. This is the hex number **F** in the left most column of Fig. 17. There are 16 possible opcodes but only 8 are implemented in MU0. The meanings of the instructions are also described in Fig. 17. The remaining 12 bits in the instruction word is the address in memory of either the operand that the instruction operates on (in the case of LDA, ADD and SUB) or the destination of the data in the ACC (STO) or the address of the next instruction in the case of the JUMP commands (JMP, JGE, JNE).

The most important thing to notice is the Register Transfer Level (RTL) design of MU0. There is a simple two state state machine which is a sequential device. The FSM controls the operation of the datapath (ALU, PC, IR, MUXes, memory... etc) by a set of combinational output voltage control levels (PCen, Wen, Ren,... etc. Notice that some of the datapath blocks may be combinational such as the MUXes and some sequential such as the PC, IR and the memory. The ALU could be either combinational or sequential.

Fig. 19 shows the instruction word format.

Figure 19: MU0 instruction format.

The program counter is incremented every instruction. It is only controlled by the active edges of the clock. Consequently MU0 automatically runs sequentially through the addresses in memory. Reading an instuction occurs when that instruction appears in the IR in the EXEC state.

The cunning in the design of the MU0 architecture is that each hardware block in the datapath of Fig 18 is configured to execute these instructions by appropriate changes in their control inputs. Examples of controls are **PCen (enable PC), ACCen (enable the ACC), Asel (choose the input that connects to the output of the address MUX, a-mux), M (choose the function to be performed by the ALU), etc**. The bit values of these controls are the outputs of the **control path FSM**.

A.2 The Control Path Finite State Machine

The **next state diagram** of the controller in Fig. 20 describes how the FSM works. The first column shows the states of which there are just two,

0 **FETCH** (fetch instruction and store instruction in IR) and,

1 **EXEC** (decode instruction in the IR and execute instruction).

The second column shows the opcode, F. The opcode is MU0's only input. MU0 obtains the opcode when the controller FSM reads bits [15:12] of the IR. The third column is the next state that the controller jumps to. Note that the opcode is not needed in the FETCH state (**0**) because in this state the only steps are to mux the PC contents onto the address bus through the a-mux and to set up the ALU input control, M, for a PC increment. As a result, regardless of the F or opcode value, the next state is EXEC (**1**). This explains the dont cares, "XXX" in the F cell in the table.

FSM State Transition Table

state	F[2:0]	Next state	IREn	PCEn	AccEn	M[1:0]	Xsel	Ysel	Asel	Ren	Wen
0	xxx	1	1	1	0	10	0	x	0	1	0
1	000	0	0	0	1	00	0	1	1	1	0
1	001	0	0	0	0	xx	1	x	1	0	1
1	010	0	0	0	1	01	1	1	1	1	0
1	011	0	0	0	1	11	1	1	1	1	0
1	100	0	0	1	0	00	0	0	x	0	0
1	101	0	0	$\overline{N}$	0	00	0	0	x	0	0
1	110	0	0	$\overline{Z}$	0	00	0	0	x	0	0
1	111	1	0	0	0	xx	0	x	x	0	0

Notes:

- N and Z are the Negative and Zero state of the Accumulator, respectively. (used to reduce the size of the table, as drawn)
- If a value is not going to be latched it doesn't matter what it is! (e.g. ALU output for STO)
- STP operates by remaining in its evaluation state.

Figure 20: MU0 next state diagram

If you look at the next-state diagram you should be able to confirm the interconnects of MU0 in Fig. 21 for the FETCH state. The grey tracks indicate connected paths in the datapath.

A.3 MU0 in action

Try and follow the following verbal description. Remember that all registers (PC, IR and ACC) and state transitions occur on the positive edge of the clock but memory read and writes occur on negative clock transitions.

The sequence of events that occur in the FETCH state from the first postive transition of the system clock are as follows.

1. The FETCH cycle occurs at the first positive edge of the clock.
2. In the FETCH state the a-mux input is connected to the PC output. The MUX is a combinational device and so the PC contents should already be pointing to the address of the next instruction in memory.

3. At tbe ensuing negative clock transition the memory transfers the contents of the location whose address is in the PC to the Dbus. The Dbus is the output data line of the memory whereas the Xbus is the input data line.

4. The contents of the Dbus are now present at the input to the IR.

5. In the FETCH state the PC contents are also pointing at the ALU input through the x-mux. The ALU M-value is set so that the ALU increments the value on this input. Since both the x-mux and the ALU are combinational devices, the PC incremented contents are transferred instantaneously at the PC input. On the next positive clock transition the contents of the PC will be incremented ready for the next time the FSM is in the FETCH state.

From the second positive clock transition we are in the EXEC state. The sequence of events that occur in the this state are as follows.

1. At this transition the PC increments its contents as discussed previously.

2. The IR registers the contents of the Dbus to its output.

3. The controller reads the [15:12] bits (the opcode) from the IR.

4. Depending on the opcode value, several function controls in the datapath may be enabled or disabled as follows.

    - If the opcode is LDA then the ACC is enabled and the y-mux is set so that the Dbus is connected to the ALU input on the Ybus. The ALU M value is set for a through connnection on its Ybus input. At the positive edge of the next clock transition into the FETCH state, the ACC output will store the contents of the Dbus.
    - If the opcode is for ADD or SUB then the ACC is enabled and the Dbus is again connected to the ALU via the Ybus through y-mux. The x-mux is set to allow the contents of the ACC onto the Xbus and the ACC M value is set for ADD or SUB. On the subsequent negative clock transition the contents of the memory is transferred onto the Dbus. At the positive edge of the next clock transition into the FETCH state, the ACC output will store the sum of its previous value and that in the memory location.
    - If the opcode is STO, the x-mux places the contents of the ACC on the input to the Xbus which is also the memory input data line. The last 12 bits of the contents of the IR are sent via the a-mux to the address bus of the memory. On the next negative clock transition the memory stores the contents of the Xbus (the contents of the ACC).

- In the case of the JUMP instructions, the last 12 bits of the instruction register are sent via the y-mux to the Ybus and the ALU. The ALU is set for straight through so that this new memory address is fed to the PC. At the ensuing posedge of clock (FETCH) the PC is changed to the address which is the operand of the JUMP instruction.

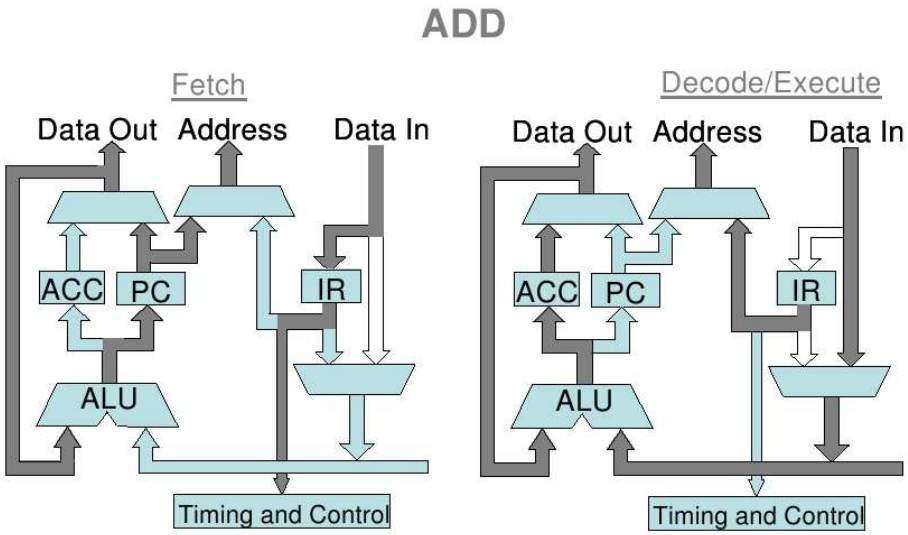


Figure 21: The MU0 datapath interconnects during FETCH and EXEC.

A.4 Running a Program on MU0

The following is a machine code listing of a MU0 program which adds the contents (000A) of memory location 4 to the contents (0001) of memory location 5. The first hex digit in each command is the opcode. These are 0 (LDA), 2 (ADD), 1 (STO). The remaining 3 hex digits are the address operands as discussed above.

```
0004  (load (LDA) the contents of memory adddress 4 into the ACC)
2005  (add (ADD) the contents of memory address 5 to that in the ACC)
1006  (store (STO) the contents of the ACC in memory location 6)
7000  (STOP)
000A  (data stored in memory location 4)
0001  (data stored in memory location 5)
0000  (data stored in memory location 6)
```

Notice how execution occurs in purely sequential fashion. MU0 does not know which memory addresses contain instructions and which data. Its proper operation depends

entirely on proper programming and the march of the PC contents. The STOP command terminates execution and prevents the processor from trying to perform a false opcode in the first hex digit of the data at memory location 4.

Fig 22 shows the complete GTKWAVE output from running MU0 with ICARUS VERILOG.

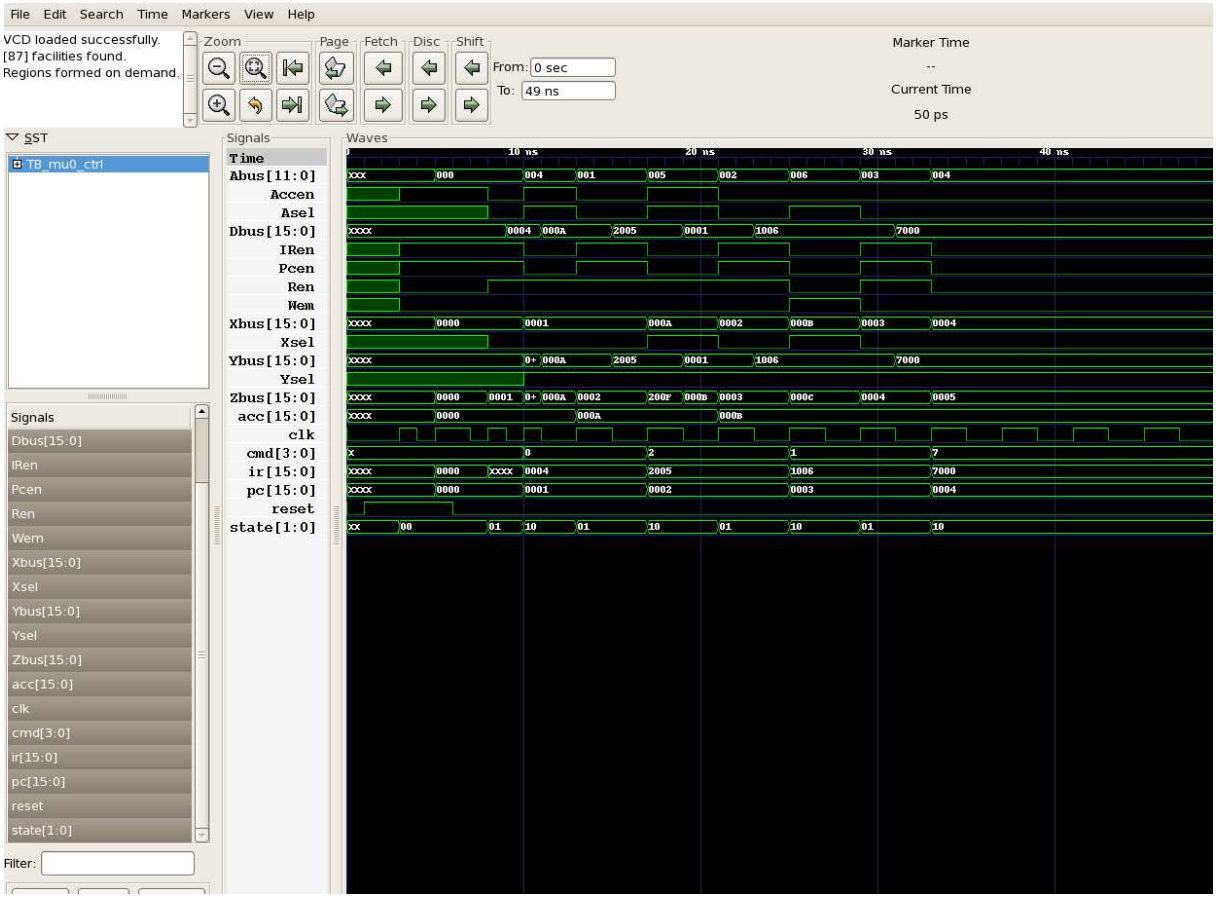


Figure 22: GTKWAVE traces of the MU0 data during execution of the above program

Fig. 23 expands the traces around the FETCH and EXEC states when the instruction 2005 is being executed. For instruction 2005 the PC is pointing to address 1 in memory. During this instruction the contents of memory address 5 (0001) is added to the contents of the accumulator which is by now 000A. Notice that the actual instruction 2005 does not appear in the IR until the EXEC state is reached and that the contents of the ACC do not register the sum, 000B, until the FETCH cycle of the following instruction.

A.5 MU0 Assembly Language?

The lexical commands in Fig. 17, LDA, STO, etc are referred to as assembly language

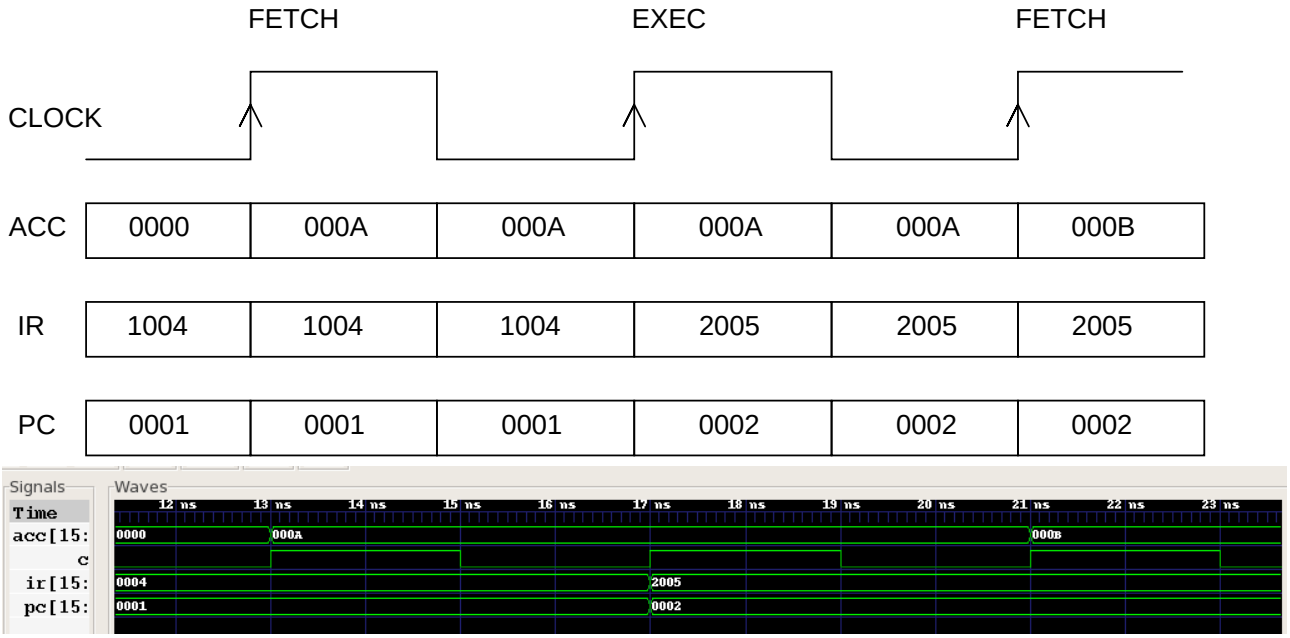


Figure 23: Expected and GTKWAVE traces of MU0 ACC, IR and PC registers around the execution of the 2005 instruction

instructions. Normally when writing programs for a microprocessor one only has to use these commands and some variables representing the data. This is clearly much easier to read than the column of numbers that form the machine code. However use of assembly language presumes the existence of an assembly language compiler or assembler for short which translates tbe assembly language into machine code. Unfortunately (to the best of my knowledge) MU0 does not have an assembler written for it (though you may be attempted to write one in JAVA or C, I dont think it would be difficult). We have already seen a little assembly language with PICOBLAZE. and later in the course we will see some more.

B Frequently Asked Questions

1. Am I allowed to use Xilinx IP-Cores and code from the NET in the project? Yes. You may use them for the FIFO if you use one.

2. What version of ISE will the RPC tests use.

   ISE WebPACK Version 13.3i

3. Concerning Fixed-point. How do you want us to enter values? If we left-justify everything as the HP-35 did, then what happens if someone enters "1.2"? Should the point be moved (so it's no longer in the middle, but the number remains left-justified) or should the display automatically change to show "01.2"?

   Fixed point in the doc spec says XX.XX. When you press "2.1" you get "02.10" on the display. When you press "2 ENTER" you get "02.00" on the display. The decimal point key would need to be used, as described in the project specification.

4. You mentioned earlier that if someone does a calculation on the one you wrote and then presses a number key, the result of the previous calculation is lost. HP automatically push that value onto the stack instead. Which one do you want?

   Since the project specification allows for the enter key to be pressed to chain calculations (the direct algorithm), the HP algorithm with digit induced stack pushing will not be tested.

5. Leading/trailing zeros, depending on left or right justification. Mine always displays trailing zeros in floating-point mode (so 25 will be shown as 25.00) and both trailing and leading zeros in fixed-point mode (so 2.5 gets shown as 02.50).

   This is fine, as long as the specified precision is displayed.

6. Rounding. Should the calculator perform rounding in any specific way?

   Not specified, not tested.

7. Minimum stack size. The HP-35 had four including the Key Holding Register (and one more for permanent storage). The size required for testing needs to be defined, as well as information on whether that size includes the Key Holding Register/accumulator.

   This has now been specified. You must use greater than or equal to four levels in addition to any Key Holding Register. For example the following must give the result 150.

   50 ENTER 40 ENTER 30 ENTER 20 ENTER 10 + + + +

8. Do we have to handle negative zero in any particular way?

   Once you derive zero its sign is irrelevant.

9. 0/0 handling. Does 0/0 = 1 (because anything divided by itself is 1), 0 (because 0 divided by anything is 0), or error?

   0/0 does not equal 1. Division by zero not be tested.

10. Additional key presses once the Key Holding Register (if implemented) is full. Should typing 1234567890 result in only 1234 being stored, or only 7890 being stored, or an error?

    Not specified, not tested.

11. Key repeats. If a key is held down for a few seconds, should that fill up the display as it does on a computer? Or should it just enter the key once? My HP-38G only enters a number once for each key press, so I've followed that.

    One digit per key press.

12. Precision. Binary calculators will lose precision with certain values (eg. 0.01 in base 10). Exactly what precision is required?

    This is explicitly specified by the project document.

13. "Unknown" key presses. A lot of people seem to be planning to implement multiply/divide in the integer calculator. If the user sends a "/" or "*" key to the calculator, should it do nothing at all (ie unknown key) or perform the operation? If the user just follows the exact assignment specification then it should do nothing because the assignment doesn't require multiplication or division, but it seems silly to ignore perfectly good operations.

    Either would be fine. But it will not be assessed.

14. Does the 'clear stack' key remove errors, or can that only be done with the reset button?

    Error flags are not specified in the project document because error detection is not necessary.

15. I heard that execution speed would be tested, but I can't see how.

    Execution speed is only being tested in as much as the speed specs on the PS/2 protocol and the serial 9600 BAUD must be met.

16. Can I enter numbers that I cannot see on the display?

    No

17. Are we correct in thinking that since the seven segment displays only have 4 digits, all our code should only bother working with 15 bits ($2^{14} = 16384$, the lowest power of 2 greater than 9999, and one bit for the sign?)

    Not necessarily. The precision refers to the display. Thus 4.3 means up to four significant digits before the dp and up to 3 after the dp. Obviously you cannot

achieve these independently. However for fixed point you can of course as you must have two before and two after the dp under all conditions.

I think that the answer to your question depends on how you do your arithmetic. If you work in BCD you will naturally be working with more bits due to its inefficient representation . e.g. 0 (base 2) = 0000 (BCD).

Division and multiplication will have other implications.

This is an important aspect of the project and I am supposing that projects will exhibit signifcant creativity and science in the solutions. I cannot wait to see your answers!

18. Are we meant to hand in separate code for the levels 1, 2, and 3? Because the fixed and floating points implementations are mutually exclusive.

    This is stated clearly in the spec. We will not search in different folders to patch together your code. You must hand in three separate sets of source code and their UCF files. This facilitates testing. The project rules make this clear and also specify that UCF files should be provided so that the projects can be built from scratch. We do not want your bit files.